Computer Applications in Engineering Education

WILEY

SHORT COMMUNICATION

Software Version Discrepancy: Arduino Spectrum Analyzer for Teaching in Data Acquisition, FFT and Real-Time System

Yoshiyasu Takefuji

Faculty of Data Science, Musashino University, Tokyo, Japan

Correspondence: Yoshiyasu Takefuji (takefuji@keio.jp)

Received: 21 August 2025 | **Revised:** 11 January 2026 | **Accepted:** 19 February 2026

Keywords: device driver inconsistency | FFT spectrum analyzer with Arduino | real-time system | software version discrepancies

ABSTRACT

Educators face significant challenges in delivering comprehensive instruction on Fast Fourier Transform (FFT) due to the scarcity of affordable, hands-on learning materials that simultaneously integrate hardware components and software applications. This paper presents an inexpensive Arduino-based real-time audio spectrum analyzer that serves as an effective educational platform for teaching data acquisition, FFT analysis, and graphical display techniques. Our implementation employs minimal components: an Arduino Nano microcontroller, an OLED 128×64 (I^2C) display, and a microphone input, creating an accessible standalone system for remote learning environments. Critically, we address the often-overlooked issue of software version discrepancies in open-source libraries, demonstrating how these inconsistencies quantifiably impact system functionality and real-time performance. Our benchmarking reveals that using incompatible library versions resulted in a 50% reduction in processing speed (from 12 fps to 6 fps), introduced a lag of more than 100 ms, and produced significant display artifacts that affected measurement accuracy. These performance differences directly impact the educational utility of the system, particularly when teaching time-critical applications. The paper provides practical guidance on calibration techniques for accurate measurement and strategies for navigating software compatibility challenges. This approach enables students and novice engineers to construct and experiment with functional spectrum analyzers outside traditional laboratory settings, while simultaneously developing crucial skills in troubleshooting the software version discrepancies that commonly affect real-world engineering projects.

1 | Introduction

In today's digitally dominated engineering curricula, hands-on analog signal processing often takes a back seat, yet understanding real-time data transforms remains crucial. This work advances engineering education by positioning low-cost Arduino platforms as bridges between analog intuition and digital signal-processing theory. Students physically capture acoustic waveforms, apply FFT algorithms [1, 2] in firmware, and immediately visualize results, closing the loop between hardware and software.

Few affordable DIY materials integrate learning Fast Fourier Transform (FFT), data capture, mathematics, and application development using both hardware and software simultaneously. Many educators lack experience with open-source FFT libraries and have never constructed analog measuring instruments, often underestimating the complexity involved in building functional spectrum analyzers. Our proposed audio FFT project fills this gap by enabling learners to experience open-source software development while mastering FFT concepts through hands-on implementation. In an increasingly digital world with dwindling numbers of analog specialists, this project provides crucial exposure to analog principles and their practical applications. Unlike software simulations, this project provides

invaluable experience with practical challenges including proper grounding techniques, real-time audio signal capture, and hardware-based FFT calculations.

This paper delivers two complementary contributions. First, it lays out the end-to-end mathematical and open-source implementation details of a 64-point, radix-2 FFT [1, 3] on the resource-constrained Arduino Nano [4] with ATmega328P microcontroller [5]. Remarkably, an Arduino Nano with just 16 MHz processing power and 32KB flash memory can perform real-time acoustic frequency measurements with FFT computation. To our knowledge, no other DIY FFT measuring instrument offers comparable capabilities at such a small scale and low cost.

We demonstrate how to select the sampling rate (≈11 kHz) to achieve a 62.5 Hz bin resolution, apply a fixed-point Hamming window via the fix_fft library to suppress spectral leakage, and optimize the in-place butterfly computations so that the entire algorithm—including data buffers and lookup tables—fits within the Nano's 2 kB of SRAM. We also describe how to stream samples from an electret microphone, synchronize ADC reads at 16 MHz CPU clock, and throttle I²C writes to an SSD1306 OLED [6] at 115200 baud to maintain real-time performance.

To validate our implementation, we built a real-time spectrum analyzer using the Arduino Nano, coupled with a microphone for audio input and an OLED 128 × 64 graphic display for visual output. The hardware processes acoustic signals through the Arduino's analog input, which are then analyzed using the fix_fft.h library. The software architecture integrates multiple components Adafruit_GFX.h provides graphics primitives, Adafruit_SSD1306.h handles display interfacing, and fix_fft.h performs spectral analysis. The computed power spectrum and amplitude values are visualized on the OLED display.

Second—and uniquely—we tackle a pervasive but underreported obstacle in open-source projects: library version conflicts. Open-source implementations require careful version control and explicit documentation of all software dependencies to ensure reproducibility and consistent performance across setups [7]. While academic and industrial communities typically recommend software updates, particularly for security purposes, the reality of software dependencies creates a more complex situation. The conventional wisdom of always updating to the latest version fails to account for the intricate relationships between software components [8]. Modern software consists of numerous interconnected library modules that may develop dependency conflicts or operational incompatibilities when updated independently [9]. In pursuing feature improvements, library designers may intentionally abandon compatibility with previous implementations, disrupting software inheritance patterns and causing system malfunctions [10]. Recent empirical studies have documented how transitive dependencies in package ecosystems can lead to cascading failures when individual components are updated without considering the broader dependency graph [11].

To assess academic recognition of this problem, we conducted targeted searches using the precise phrase "software version discrepancy" within premier scientific publication domains. Surprisingly, these searches yielded no results, suggesting that despite its practical significance, this issue remains largely unaddressed in top-tier scientific literature. A broader Google search returned only 13 documents, with just one appearing in a peer-reviewed journal [12], which lacked specific examples and practical solutions.

Through exhaustive testing of Adafruit_SSD1306 releases (v1.0.0 through v2.2.1) [6], we document how incremental driver changes can introduce I²C timing overhead, buffer overflows, or API mismatches that silently break the spectrum display or corrupt FFT input. By correlating driver version to observed frame rates and spectral fidelity, we demonstrate that "upgrading to the latest" is not always safe. The proliferation of new devices and sensors has driven continuous improvement and expansion of open-source software libraries. However, contrary to expectations, newly added functions can sometimes degrade overall system performance rather than enhance it.

For clear illustration of these issues, we encourage readers to view our two demonstration videos: one showing slow FFT performance due to software discrepancy issues, and another demonstrating normal FFT operation after resolving these compatibility problems. The "slow FFT video" utilizing the latest software library exhibits significant performance degradation—a direct consequence of version incompatibility—while the "normal FFT video" using an older software library performs optimally.

This paper elevates software-dependency governance to the same level of importance as circuit layout or algorithm design. Library consistency is not merely a technical consideration but a fundamental component of scientific reproducibility in open-source ecosystems. Our empirical investigation demonstrates that students and system designers must develop a nuanced understanding of library versioning and inter-library consistency while maintaining real-time performance requirements.

We propose a workflow of controlled version pinning, combinatorial driver testing, and environment snapshots to guarantee reproducible, classroom-ready builds of real-time audio spectrum analyzers, establishing a new "compatibility-first" paradigm for engineering education. This implementation not only creates a functional measurement device but also provides an accessible platform for students to explore FFT mathematics through open-source code, bridging the gap between theoretical concepts and practical applications.

2 | Methods

We evaluated version-related discrepancies in the Adafruit_SSD1306 library across releases from v1.0.0 through v2.2.1. Using a real-time audio spectrum analyzer as the test application, we identified a consistent lag exceeding 100 ms in certain versions, along with performance and rendering differences attributable to driver updates, buffer handling, and I²C transaction timing.

2.1 | Hardware

The spectrum analyzer comprises an Arduino Nano paired with an amplified electret microphone module (5 V, OUT, GND) and a 128 × 64 OLED I²C display. The microphone module used is either GY-MAX4466 or SparkFun Electret Microphone Breakout. Figure 1a presents the circuit schematic, while Figure 1b shows the breadboard implementation. The wiring consists of

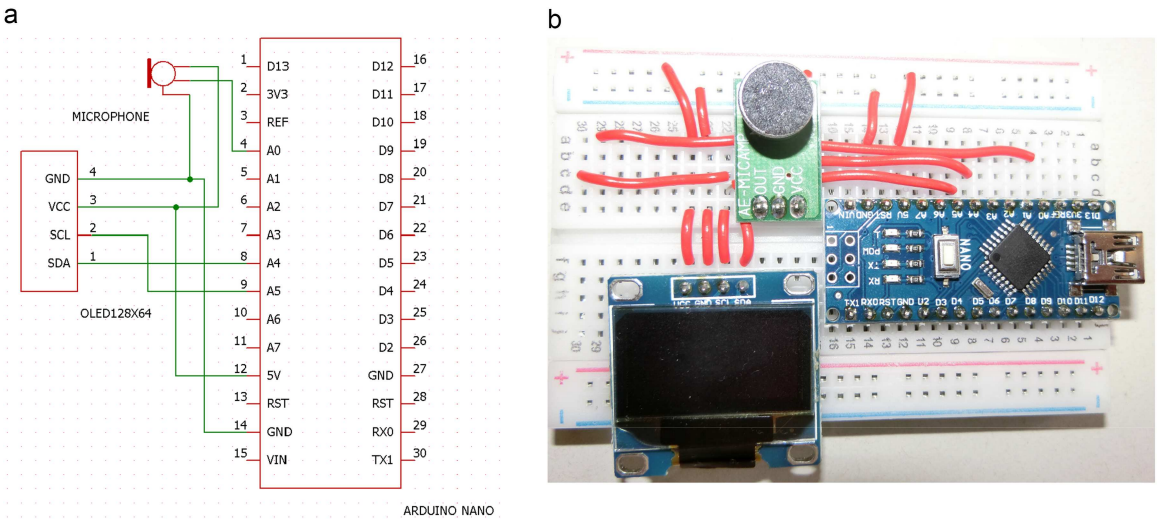

FIGURE 1 | a Circuit diagram of spectrum analyzer. b Implemented spectrum analyzer. Supplement: f_est. py import pandas as pd import statsmodels. api as sm import matplotlib. pyplot as plt data=pd. read_csv('f. csv') x=data['num'] x=sm. add_constant(x) y=data['f'] est=sm. OLS(y,x). fit() print(est. params) print(est. rsquared) print(est. rsquared. astype(float). round(6)) e=est. predict(x). astype(int) x=data['num'] plt. scatter (x,y,c='red') #plt. scatter(x,e,c='blue') plt. xlabel('index') plt. ylabel('frequency') plt. scatter(x,x*71-7,c='black') from sklearn. metrics import r2_score print(r2_score(y,x*71-7)) plt. show().

I²C connections (SDA, SCL) between the OLED and the Arduino Nano, and an analog input connection from the microphone output to pin A0, with shared 5 V power and ground rails for stable operation.

2.2 | Software

Figure 2 shows the main Arduino sketch (fftOLED_peak. ino) targeting an Arduino Nano (ATmega328P). The full source is available on GitHub [13].

To build and upload the sketch, use Arduino IDE 1.8.19 (or later). After installation, open Tools → Board → Arduino AVR Boards and select Arduino Nano (ATmega328P, 16 MHz, 5 V). Choose the appropriate COM port under Tools → Port and set the upload speed to 115200 baud. These settings align ADC sampling, display refresh, and serial logging for reliable real-time performance.

Manage library dependencies via Sketch → Include Library → Manage Libraries. Install:

- Adafruit_GFX (e.g., v1.10.13)
- Adafruit_SSD1306 (tested across v1.0.0 to v2.2.1)
- fix_fft (for FFT and optional windowing)

For comprehensive driver testing, download each Adafruit_SSD1306 release into your sketchbook's libraries directory (<sketchbook > /libraries), renaming folders to maintain multiple versions side by side (e.g., Adafruit_SSD1306_v1_0_0, Adafruit_SSD1306_v2_2_1). This approach allows rapid switching and recompilation to assess how each driver version affects I²C timing, display-update latency, and overall analyzer responsiveness.

For FFT frequency testing, the Windows binary executable wg. exe (waveform generator) was employed to validate testing frequency [14]. We tested a single primary FFT frequency and logged all audio and video data during experiments.

3 | Results

The Adafruit_SSD1306 device driver at version 2.2.1 (as of May 1, 2020) noticeably degrades the performance of the proposed spectrum analyzer, introducing measurable display-update latency and reduced responsiveness. In contrast, the earliest release series shows more favorable behavior: versions 1.0.0, 1.0.1, 1.1.0, and 1.1.2 consistently deliver a reasonable real-time response in our tests. These differences are likely tied to changes in buffer management, I²C transaction patterns, and drawing routines introduced in later releases.

A real-time demonstration of the spectrum analyzer operating without perceptible lag is available [15]. A contrasting demonstration showing more than 100 ms of lag can be viewed [16]. The reported values (100 ms lag and 6–12 fps) were obtained from analysis of these recordings. Two experiments conducted with different library versions demonstrated a consistent performance difference, with the slower implementation exhibiting a 100 ms latency compared to the fast real-time implementation.

The measurable frequency range of the analyzer spans approximately 300 Hz to 4500 Hz. Frequency accuracy was validated. To quantify measurement fidelity, we compared the analyzer's measured spectrum peaks against known input tones. With wg. exe on a Windows PC, we generated single-frequency sine waves and recorded the corresponding FFT peak index (num) from the Arduino IDE Serial Monitor for calibration. Test tones included 550 Hz, 630 Hz, 800 Hz, 1000 Hz, 1600 Hz, 3000 Hz, 4000 Hz, and 4500 Hz. The resulting frequency–index pairs were saved to f. csv for subsequent analysis.

For model fitting, we applied ordinary least squares (OLS) via a Python script (f_est. py, Supplement) to derive a linear calibration between frequency and FFT index using the measured f. csv. Install Python 3.8 (or 2.7) with miniconda to run the script. The regression produced:

```
/*
  * Arduino Real-time Spectrum Analizer
  * install Adafruit_GFX library
  * install Adafruit_SSD1306 library: verson 1.0.0 ~ 1.1.2 device driver
  * install fix_fft library
  */
#include <fix_fft.h>                    //library to perfom the Fast Fourier
Transform
#include <Wire.h>                                   //I2C library for OLED
#include <Adafruit_GFX.h>                          //graphics library for
OLED
#include <Adafruit_SSD1306.h>                       //OLED device driver

#define OLED_RESET 4                    //OLED reset
Adafruit_SSD1306 display(OLED_RESET);                //declare instance

char im[128],data[128],lastpass[64];                 //variables for the
FFT
char x = 0, ylim = 60;                             //variables for drawing the
graphics
int i = 0, val;                                     //counters

void setup()
{
   Serial.begin(9600);
   display.begin(SSD1306_SWITCHCAPVCC,0x3C);          //begin OLED
   display.setTextSize(1);                          //set OLED text size to 1 (1-6)
   display.setTextColor(WHITE);                       //set text color to white
   display.clearDisplay();                          //clear display
   analogReference(DEFAULT);                        // Use default (5v)
voltage.
   for (int z=0; z<64; z++) {lastpass[z]=80;};
};

void loop()
{
```

FIGURE 2 | Sketch of fftOLED_peak. ino.

```
   int min=1024, max=0,high=0,index=0;        //set minumum & maximum ADC
values
   for (i = 0; i < 128; i++) {                              //take 128 samples
      val = analogRead(A0);                                  //get audio from Analog
0
      data[i] = val / 4 - 128;                             //each element of array is val/4-
128
      im[i] = 0;                                                //
      if(val>max) max=val;                                  //capture maximum level
      if(val<min) min=val;                                  //capture minimum level
   };

   fix_fft(data, im, 7, 0);                                 //perform the FFT on data

   display.clearDisplay();                                  //clear display
   for (i = 1; i < 64; i++) {                          // In the current design, 60Hz and
noise
     if(i!=41) data[i] = sqrt(data[i] * data[i] + im[i] * im[i]);
     if(data[i]>high) {high=data[i];index=i;};
     display.drawFastVLine(2*i, 2,lastpass[i],WHITE);
     lastpass[i]=30-data[i];

   };
   Serial.print(high,DEC);
   Serial.print(" ");
   Serial.println(index,DEC);
   display.setTextSize(2);

display.setTextColor(WHITE);
display.setCursor(5,17);
if(index>4&& index!=41)display.print(index*71-7);
display.print(" Hz");
display.display();                                          //show the buffer
};
```

FIGURE 2 | (Continued)

$$\text{const} = -6.535112$$
$$\text{num} = 70.755618$$
$$\text{r} - \text{squared} = 0.999634$$

These coefficients yield the calibration formula $f = -6.535 + 70.756 \cdot \text{num}$ (rounded). For simplicity in real-time use, we adopt the near-equivalent approximation $f = -7 + 71 \cdot \text{num}$, where num is the FFT peak index and −7 is the intercept. Figure 3 illustrates the alignment between real and estimated frequencies versus index. The high coefficient of determination ($R^2 = 0.999634$) confirms excellent linearity and estimation accuracy across the tested range.

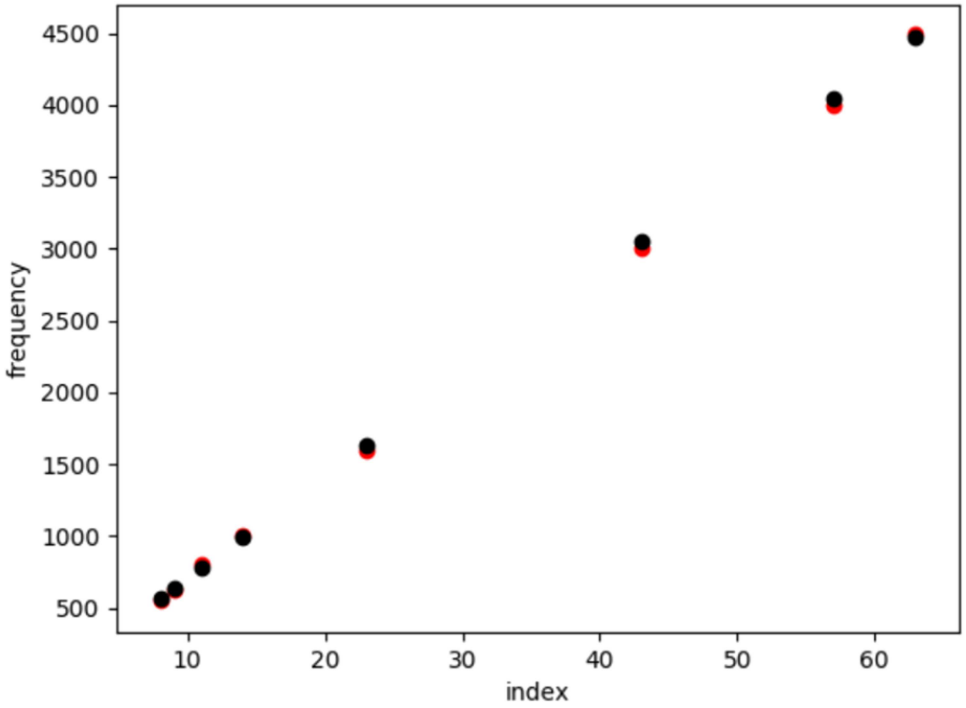

FIGURE 3 | Relationship between frequency and FFT index number (real: red, estimated: black).

With the correct software version, the audio spectrum achieves roughly 80% accuracy (±10%), operates at 6–12 fps, and maintains latency under 50 ms; by contrast, an incorrect driver version introduces more than 100 ms of lag, degrading accuracy and responsiveness.

4 | Discussion

Our work advances low-cost, hands-on signal-processing education by unifying data acquisition, FFT computation, and real-time visualization on a minimal Arduino platform. Conceptually, it demonstrates how open-source hardware and software can be leveraged to teach the full signal-processing pipeline—from analog sensing through digital transformation to graphical output—while exposing students to critical engineering trade-offs. The approach also highlights how subtle implementation choices (e.g., buffer size, I²C transaction batching, and drawing primitives) materially affect end-to-end latency and user perception of "real time."

In testing software compatibility, we found that newer 'Adafruit_SSD1306' drivers (v2.2.1) introduce I²C latency and buffer overhead that effectively halve display update rates, causing sample overruns and spectral smearing. Earlier releases (1.0.0–1.1.2) yielded stable real-time performance, likely due to leaner buffer management and simpler drawing routines. This counterintuitive result emphasizes the need for exhaustive combinatorial testing across driver versions, board cores, and compiler settings. It also argues for disciplined reproducibility practices—Git tagging of known-good versions, locked dependency manifests, or containerized toolchains—to prevent silent regressions. Empirically, the contrast between a lag-free demonstration [15] and a > 100 ms lag case [16] underscores how display-stack changes alone can tip a system from responsive to unacceptably delayed.

Our calibration with eight known sine tones yielded a near-linear mapping, $f = -7 + 71 \cdot \text{index}$ ($R^2 = 0.999634$), closely matching the OLS fit from 'f. csv' using 'f_est. py' (const = −6.535, num = 70.756). The small negative intercept plausibly reflects cumulative timing offsets (ADC start-up, ISR scheduling) and microphone low-frequency roll-off, while the slope's deviation from an ideal bin width (e.g., 62.5 Hz/bin in some canonical setups) points to practical factors such as effective sampling rate, jitter, and window leakage. Beyond producing a usable real-time frequency readout, the exercise provides a concrete, data-driven path for students to explore model fitting, hardware characterization, and uncertainty—connecting measurement to theory in a way that is immediately observable on the device.

Nonetheless, the analyzer's usable band (≈300–4500 Hz) remains bounded by the electret capsule response, ADC throughput, and Nyquist constraints, while fixed-point FFT limits spectral resolution and dynamic range. Manual library management further adds development overhead and invites variability. Future work will target bandwidth extension via pre-filtering or higher-order anti-alias filters, lower-latency display paths through optimized I²C clocking or DMA-driven screens, and improved spectral fidelity using alternative window functions and higher-bit-depth DSP. To harden reproducibility, we plan an automated library-dependency validator and Docker-based images that freeze toolchain and driver versions, ensuring that the demonstrated real-time behavior is portable and durable across environments.

5 | Conclusion

Depending on the installed display driver, the spectrum analyzer's behavior can shift from responsive to laggy—an outcome we directly observed with 'Adafruit_SSD1306' v2.2.1 versus the

earlier 1.0.0–1.1.2 series. To ensure reproducibility, projects like this must declare and lock all software library versions and toolchain details (driver release, board core, compiler, and I²C settings). Versioned dependencies, tagged repositories, and containerized builds are not optional niceties; they are essential controls that prevent silent performance regressions and make results portable across student setups.

Despite these challenges, the proposed low-cost platform successfully unifies microphone-based data acquisition, open-source FFT computation, and OLED visualization, providing a complete, hands-on signal-processing pipeline. It also offers a practical lesson in the consistency—and inconsistency—of open-source libraries: students see firsthand how buffer management and I²C transaction changes can alter real-time performance, and they learn how to diagnose and remedy such issues through disciplined version management.

This work demonstrates that real-time FFT instruction can be delivered effectively via online lectures using inexpensive parts on a breadboard with single-core wires, enabling at-home practice during disruptions such as the COVID-19 pandemic. Learning FFT on a standalone system becomes a capstone exercise: capturing audio, displaying spectra on an OLED, recognizing software-version discrepancies, and applying fixes (pinning library versions, validating performance, and documenting configurations). With minimal hardware and readable source code, students can replicate the system, perform spectrum analysis at home, and develop robust habits for building reproducible, real-time embedded applications with open-source tools.

Author Contributions

Yoshiyasu Takefuji completed this research and wrote the program and this article.

Funding

The author received no specific funding for this work.

Ethics Statement

The author has nothing to report.

Consent

The author has nothing to report.

Conflicts of Interest

The author declares no conflicts of interest.

Code Availability

All program codes are publicly available at GitHub. https://github.com/y-takefuji/IoT/blob/main/fftOLED_peak.ino.

Data Availability Statement

The author has nothing to report.

References

1. P. Ray, N. Parida, S. Biswal, and A. R. Singh, "Development of a Power Quality Analyzer Using Arduino Technology," in *Recent Advances in Power Electronics and Drives*. Lecture Notes in Electrical Engineering, eds, S. Kumar, B. Singh, V. K. Sood, vol 973 (Springer, 2023), https://doi.org/10.1007/978-981-19-7728-2_17.

2. L. Ashok Kumar, V. Indragandhi, R. Selvamathi, V. Vijayakumar, L. Ravi, and V. Subramaniyaswamy, "Design, Power Quality Analysis, and Implementation of Smart Energy Meter Using Internet of Things," *Computers & Electrical Engineering* 93 (2021): 107203, https://doiorg/10.1016/j.compeleceng.2021.107203.

3. Kosme. (2020). FFT Library. GitHub, https://github.com/kosme/fix_fft.

4. A. Dabbous, M. Fresta, F. Bellotti, and R. Berta, "Arduino Nano-Based System for Tennis Shot Classification," in *Proceedings of SIE 2023*, eds. C. Ciofi and E. Limiti. (Springer, 2024), 357–362, https://doi.org/10.1007/978-3-031-48711-8_43.

5. N. Dunbar, "ATmega328P Configuration and Management," in *Arduino Software Internals*. Maker Innovations Series (Apress, 2024), 219–257, https://doi.org/10.1007/979-8-8688-0171-6_7.

6. T. Ching, J. Vasudevan, H. Y. Tan, et al., "Highly Customizable 3D-printed Peristaltic Pump Kit," *HardwareX* 10 (2021): e00202, https://doi.org/10.1016/j.ohx.2021.e00202.

7. N. Ahmad and N. Tripathi, "Benefits, Challenges, and Implications of Open-Source Software for Health-Tech Startups: An Empirical Study," in *Software Business. ICSOB 2023*. Lecture Notes in Business Information Processing, eds. S. Hyrynsalmi, J. Münch, K. Smolander, and J. Melegati, Vol. 500 (Springer, 2024), https://doi.org/10.1007/978-3-031-53227-6_19.

8. A. Decan, T. Mens, and P. Grosjean, "An Empirical Comparison of Dependency Network Evolution in Seven Software Packaging Ecosystems," *Empirical Software Engineering* 24 (2019): 381–416, https://doi.org/10.1007/s10664-017-9589-y.

9. E. Wittern, P. Suter, and S. Rajagopalan, 2016, " A Look at the Dynamics of the JavaScript Package Ecosystem," in *ACM 13th Working Conference on Mining Software Repositories (MSR)*. (IEEE, 2016), 351–361, https://doi.org/10.1145/2901739.2901743.

10. S. Raemaekers, A. van Deursen, and J. Visser, "Semantic Versioning and Impact of Breaking Changes in the Maven Repository," *Journal of Systems and Software* 129 (2017): 140–158, https://doi.org/10.1016/j.jss.2016.04.008.

11. A. Zerouali, E. Constantinou, T. Mens, G. Robles, and J. González-Barahona, "An Empirical Analysis of Technical Lag in npm Package Dependencies." in *New opportunities for software reuse: ICSR*, eds. R. Capilla, B. Gallina, and C. Cetina, Vol. 2018 (Springer, 2018), 95–110, https://doi.org/10.1007/978-3-319-90421-4_6.

12. D. Lee and N. Park, "Technology and Policy Post-Security Management Framework for Iot Electrical Safety Management," *Transactions of the Korean Institute of Electrical Engineers* 66, no. 12 (2017): 1879–1888, https://doi.org/10.5370/KIEE.2017.66.12.1879.

13. GitHub. (2025). fftOLED_peak.ino, https://github.com/y-takefuji/IoT/blob/master/fftOLED_peak.ino.

14. WG140.exe. (2025), http://efu.jp.net/soft/wg/WG140.ZIP.

15. YouTube. (2025a). Real-time Demonstration, https://www.youtube.com/watch?v=SJybXe6msLk.

16. YouTube. (2025b). Demonstration With More Than 100ms Lag, https://www.youtube.com/watch?v=wWtuWTPgbos.